Non-Deterministic Finite Automata

In this section we extend our definition of deterministic finite automata to a seemingly more powerful notion, that of non-deterministic finite automata.

The surprising and wonderful result of this section is that these non-deterministic automata are actually not more powerful; they in fact describe the same set of languages.

Reading

Section 1.2 (p. 47-54)

Practice problems (page 85): 1.7, 1.8, 1.9, 1.10, 1.11, 1.14, 1.16, 1.40, 1.41, 1.42, 1.51

Challenge: 1.43, 1.44

Motivation for non-deterministic automata

Finite automata have a certain rigidity to them: At every state and a given input, there is exactly one other state to transition to. This is precisely why they are called "deterministic".

But in so many practical situations we encounter non-determinism and are confronted with choices. A good example of this is trying to recognize the concatenation of two regular languages:

$$AB = \{wv \mid w \in A,\, v \in B\}$$

If we imagine a deterministic automaton trying to use the automata for A and B along the way, we could for instance imagine it starting with the automaton for A, then continuing with the automaton for B. It is this "continuing" part that is difficult: At what point should we drop A and start looking at B? How do we know this is the right time to do so?

To make this more concrete, suppose that the overall input is 1101001, and suppose that the words 11, 110 and 11010 are all valid words in A. Then that longer input may be in AB because 01001 is in B, or because 1001 is in B, or because 01 is in B, or maybe for all 3 reasons. But we can't know until we start looking into B. So after we have read the first two numbers following A's automaton, we have arrived at an accept state for A; do we continue or do we start looking into B? What if we do start at B and 0100 turns out not to be in B? We would conclude that the whole input isn't in AB (even though it could be there for other reasons).

So we have to make a choice at that point, and we don't know what the right choice would be, and we can't afford to make the wrong choice. So we can't make a choice. This is the problem presented by deterministic automata.

Definition of non-deterministic automata

The idea of non-deterministic automata is simple: We preserve the finite-ness and definite-ness of the states of a DFA, but we become more flexible on the transitions. From a state and on a given next input, you may now transition to 0 or more states. We also allow for "free transitions", called "epsilon-transitions", from a state to another without consuming any input. This way, at any given moment in the computation, our automaton might be in a variety/set of different states, not just one. And on each new input, the automaton would follow that input from all the different states it might have been in, resulting in a new list of possible states. When the computation ends, the automaton would possibly be in any number of possible states, and as long as one of these is an accepting state then the automaton would accept the string.

> A **(Non-deterministic) Finite Automaton** (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:
>
> - Q is a finite set, called the *states*,
> - Σ is a finite set, called the *alphabet*, and we use Σ_ϵ to denote the alphabet extended with a new special symbol, ϵ, to indicate no use of input,
> - $\delta\colon Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the *transition function*,
> - $q_0 \in Q$ is the *start state*,
> - $F \subset Q$ is the set of *accept or final states* (possibly empty)

Here $\mathcal{P}(Q)$ denotes the power-set of the set Q. In other words the return values of the transition function are whole sets of states, instead of individual states. We often can split the function up in two parts, one that handles the *epsilon transitions*, i.e. transitions on no input at all, and one that handles the normal transitions.

Computation with an NFA

The meaning of computation with an NFA is similar to that for a DFA, except that we have to allow for epsilon transitions. Intuitively, a string is recognized by an NFA if we can reach a final state in one of all the possible calculations that use the string as input. More formally:

> We say that an NFA recognizes the string w, if we can write $w = y_1 y_2 \cdots y_n$ where each $y_i \in \Sigma_\epsilon$, and we have a sequence of states $r_0, r_1, \ldots, r_n$ such that:
>
> $r_0 = q_0$ is the start state of the automaton, $r_{i+1} \in \delta(r_i, y_{i+1})$ is one of the possible states to transition to on each next step, $r_n \in F$ is a final state.
>
> We say that the NFA *recognizes* a language L, if it accepts exactly the strings that are in the language.

So the formal definition has to make two allowances: The insertion of "epsilon steps" in the strings/alphabet, and the fact that the result of a call to the transition function is a whole set of possible states, so the next state just has to be an element of that set.

Epsilon Closures

One concept essential to understanding DFAs is that of epsilon closures. The idea is essentially that we want to follow all possible epsilon transitions from a given set:

> The **epsilon closure** of a set of states S, denoted $E(S)$ is the set of all states that can be reached from S via following epsilon transitions. Formally, a state s is in $E(S)$ if and only if there is a sequence of states $s_0, s_1, \ldots, s_k$ such that:
>
> - $s_0 \in S$
> - $s_{i+1} = \delta(s_i, \epsilon)$ for all i
> - $s_k = s$

To compute the epsilon closure of a set, we can proceed in steps:

- Start with $S_0 = S$.
- Compute S_1 by following a single epsilon-step from all points in S_0.
- Compute S_2 by following a single epsilon-step from all points in S_1.
- Compute S_3 by following a single epsilon-step from all points in S_2.

and so on. Since there is a finite set of states, this process will eventually stabilize. We have then arrived at the epsilon closure $E(S)$.

Standard constructions

In this section we will describe in the language of NFAs how to do the standard constructions, namely:

> Given two NFAs $M_1 = (Q_1, \Sigma, \delta_1, q_0^{(1)}, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_0^{(2)}, F_2)$ that recognize the languages L_1 and L_2 respectively, we will construct:
>
> - An NFA that recognizes the union $L_1 \cup L_2$,
> - An NFA that recognizes the concatenation $L_1 L_2$,
> - An NFA that recognizes the Kleene star L_1^*.

Union

The idea is simple:

- The states of the new NFA will include all states from M_1 and from M_2, along with one new state.
- This new state is the start state.

- There are epsilon transitions from this new start state to the start states of M_1 and M_2. All the old epsilon transitions are still present.
- There are no new normal transitions. All the old normal transitions are still present.
- The set of final states is the union of the two sets of final states for M_1 and M_2.

More formally, the new NFA is the 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- $Q = Q_1 \cup Q_2 \cup \{q_0\}$ where q_0 is a new state (and also the start state).
- $F = F_1 \cup F_2$.
- $\delta(q_0, \epsilon) = \{q_0^{(1)}, q_0^{(2)}\}$.
- $\delta(q_0, e) = \emptyset$ if $e \in \Sigma$.
- $\delta(r, e) = \delta_1(r, e)$ if $r \in Q_1$.
- $\delta(r, e) = \delta_2(r, e)$ if $r \in Q_2$.

Technically we would now need to write a formal proof of why this automaton recognizes the union of L_1 and L_2, which would involve two claims:

- That every string in $L_1 \cup L_2$ is accepted by this automaton.
- That no other strings are accepted by this automaton. Or via the contrapositive, every string that is accepted by the automaton is in the union $L_1 \cup L_2$.

The proof in many of these "constructed" cases is straightforward but tedious. We will first describe it here:

For the first part: If a string is in L_1 or in L_2, then we can take the path that the string would follow in the corresponding automaton M_1 or M_2, and prepend the epsilon transition to it from q_0 to the appropriate start state. This would show that the new automaton accepts all strings from the union.

Conversely, for the second part, suppose that we have a string that is accepted by our constructed automaton. Then that string must start with an epsilon transition from q_0, and the rest of it is essentially a path that we could have traced in L_1 or L_2, depending on where that first step took us. Since there are no transitions from states in M_1 to states in M_2, a path that arrives at a state in F and that got into M_1 to begin with must in fact end up in F_1. In other words, if that string is accepted for the union automaton, it must also be accepted for either M_1 or M_2.

Now we will make a more formal proof.

We start with the one direction. If a string 2 is in say L_1 (the same argument would work for L_2) we want to show it is accepted by our automaton. This would show that our automaton accepts every string in $L_1 \cup L_2$. According to the definition of acceptance, we must be able to write $w = y_2 y_3 \cdots y_k$, where each y_i is an ϵ or in the alphabet, and there are states r_1, r_2, r_3 and so on from Q_1 such that:

- $r_1 = q_0^{(1)}$

- $r_{i+1} = \delta_1(r_i, y_{i+1})$
- $r_k \in F_1$ is a final state for M_1

Then if we consider writing $w = \epsilon y_2 y_3 \cdots y_k$, and the states $q_0, r_1, r_2, \ldots$, then this shows that our automaton accepts the string w:

- $q_0 = q_0$ is the start state.
- $r_1 = \delta(q_0, \epsilon)$ by the added epsilon transitions from q_0.
- $r_{i+1} = \delta(r_i, y_{i+1})$ (since it is in fact just $\delta_1(r_i, y_{i+1})$)
- $r_n \in F$ since it is in F_1 and $F = F_1 \cup F_2$.

This is only one of the directions. Now we need the opposite direction. Namely that if our automaton accepts the string w, then w must be in $L_1 \cup L_2$.

Since our automaton accepts the string w, we can write $w = y_1 \cdots y_k$ and there are states r_0, r_1, . . . , r_k such that:

- $r_0 = q_0$.
- $r_{i+1} = \delta(r_i, y_{i+1})$.
- $r_k \in F$.

Since $r_1 = \delta(q_0, y_1)$ and the only transitions out of q_0 are epsilon transitions, it follows that $y_1 = \epsilon$ and that r_1 is either $q_0^{(1)}$ or $q_0^{(2)}$. The argument is very similar in either case, so let's assume that $r_1 = q_0^{(1)}$ is the start state of M_1, the automaton for language L_1. Since the transition function for states that came from M_1 is exactly δ_1, it follows that all the states r_i from r_1 and on must be in Q_1. In particular, $r_k \in F$ must also be in Q_1, and hence must be in F_1, a final state for M_1. Then since $y_1 = \epsilon$, the expression of $w = y_2 \cdots y_k$ and the states r_2, r_3, . . . , r_k show that w is accepted by M_1, and hence it is necessarily in L_1. This is what we were trying to prove.

So now you know what a "formal proof" looks like. It is worth practicing these in the examples that follow.

Concatenation

Concatenation is equally straightforward: Start with the automaton for L_1, and for each final state there add an epsilon transition to the start state of L_2.

More formally, the new NFA for the concatenation is the 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- $Q = Q_1 \cup Q_2$.
- $q_0 = q_0^{(1)}$.
- $F = F_2$.
- $\delta(r, \epsilon) = \delta_1(r, \epsilon) \cup \{q_0^{(2)}\}$ for all $r \in F_1$.
- $\delta(r, x) = \delta_1(r, x)$ for all $r \in Q_1$ (except if $r \in F_1$ and $x = \epsilon$ where the previous item applies).
- $\delta(r, x) = \delta_2(r, x)$ for all $r \in Q_2$

Exercise for the students: Write proof formally that this recognizes the language AB.

Kleene Star

The idea for the Kleene star is also straightforward. We would effectively want, first of all, to make the start state an accept state. The problem is that that state may be reachable not only on an empty string, but perhaps after a number of other transitions. Turning it into an accept state will alter the set of strings accepted by the automaton.

We therefore need to add a new start state, with an epsilon transition to the old start state. We also need to add arrows from the final states to this new start state (or they could also go to the old start state actually). This allows the automaton to try to match another substring from L_1, after it's completed matching one (hence the "star" allowing for an arbitrary number of words from A concatenated together).

Exercise: Show by an example how not introducing a new start state does not produce the correct language.

Let's try to define it more formally (starting with the automaton $M_1 = (Q_1, \Sigma, \delta_1, q_0^{(1)}, F_1)$ that recognizes L_1:

- $Q = \{q_0\} \cup Q_1$.
- $F = \{q_0\} \cup F_1$.
- $\delta(q_0, \epsilon) = \{q_0^{(1)}\}$.
- $\delta(q_0, x) = \emptyset$.
- $\delta(r, x) = \delta_1(r, x)$ for any $r \in Q_1$ that is not in F_1.
- $\delta(r, x) = \{q_0^{(1)}\} \cup \delta_1(r, x)$ for any $r \in F_1$.

The DFA equivalent to an NFA

It should be pretty clear that any DFA can be considered as an NFA. But take a moment to work out the details.

What this means is that all regular languages are recognized by some NFA (since they are recognized by a DFA and that DFA has an equivalent NFA).

> We say that two DFAs/NFAs are **equivalent**, if they recognize the same language.

Every DFA has an equivalent NFA. The remarkable fact is that the converse is also true:

> For each NFA there is a DFA that recognizes the same language.

This sounds quite remarkable: At a first glance it would seem that NFAs have the potential to recognize more languages. After all, they have all this nondeterminism and can follow multiple paths at the same time. This should be giving them more potential! In fact it does not.

Let us think about the nondeterminism for a second. First of all, from a state and on a given input the NFA might transition into a number of different states. And to top it all off, we can have epsilon transitions from those states to some other new states. So at any given time in the computation, the NFA may be in a number of different states. How can a DFA possibly do something similar?

The idea is simple: Make that whole set of possible states be one of the states on the DFA. Then on a new input letter, consider from each of these possible states in the NFA all the places that input letter could take you, following epsilon transitions when possible. Then create a new set of all the possible states. That's perhaps another state in the DFA.

Construction

So this is how the construction goes. We will do a general case first, then look at how we can make it "smaller". We start with an NFA $M_N = (Q_N, \Sigma, \delta_N, q_0^{(N)}, F_N)$. We construct a DFA $M_D = (Q_D, \Sigma, \delta_D, q_0^{(D)}, F_D)$ as follows:

- $Q_D = \mathcal{P}(Q_N)$ consists of one "state" for each possible subset of states from Q_N.
- For a set $S \in \mathcal{P}(Q_N)$, which is a state in Q_D, and an input element e, we define $\delta_D(S, e)$ as "the set of all states in Q_N that can be reached from a state in S on input e and possibly following epsilon transitions". More formally, we could define δ_D as:
$$\delta_D(S, e) = E\left(\bigcup_{q \in S} \delta_N(q, e)\right)$$
where E denotes forming the epsilon closure.
- $q_0^{(D)} = E\left(\left\{q_0^{(N)}\right\}\right)$ is the epsilon closure of the set consisting of M_N's start state.
- $F_D = \{S \in Q_D \mid S \cap F_N \neq \emptyset\}$. A state in Q_D, i.e. a set of states in Q_N, is final if and only if at least one of these states is a final state of M_N.

Question: Do we need the "empty set" state in Q_D? What does it represent? What transitions are there from and to it?

We claim that the DFA just described is equivalent to the NFA M_N. Before we prove this, note that we can make Q_D a bit smaller (as it is, if the NFA has n states then the corresponding DFA has 2^n states) as follows: We start with $q_0^{(D)}$ and construct as above all transitions from it on any possible input. This gives us a series of new states in Q_D. We do the same for each of them, until no new states are produced.

In other words, we end up including in the final Q_D only those states from $\mathcal{P}(Q_N)$ that are actually reachable from the start state.

Proof

The proof will once again consist of two directions:

- If a string w is recognized by the NFA M_N, then it should be recognized by the DFA M_D as well.
- If a string w is recognized by the DFA M_D, then it should be recognized by the NFA M_N as well.

We prove the first direction. Suppose that w is recognized by M_N. This means that there must be a way to write $w = y_1 y_2 \cdots y_k$ where the y_i are either alphabet letters or ϵ, and states $r_0, r_1, \ldots, r_k \in Q_N$ such that:

- $r_0 = q_0^{(N)}$
- $r_{i+1} \in \delta_N(r_i, y_{i+1})$
- $r_k \in F_N$

We need to turn this into an appropriate "path" in M_D. We start from $q_0^{(N)}$, and hence the DFA state $t_0 = q_0^{(D)}$, and consider y_1. If it is ϵ, this means that r_1 is in the epsilon-closure of $q_0^{(N)}$, and therefore we are still at $t_0 = q_0^{(D)}$ from the point of view of the DFA. Also since y_1 is just ϵ, we could basically ignore it as far as writing w is concerned. We then look at y_2 and continue in this manner, until we find the first non-epsilon y_{i+1}. Now what we know is that $r_i \in q_0^{(D)}$, and that $r_{i+1} \in \delta_N(r_i, y_{i+1})$, therefore r_{i+1} is one of the elements of the set that is the state $\delta_D(q_0^{(D)}, y_{i+1})$, because of the way we defined that state. We call this state t_1.

After that y_{i+1} there might be some epsilons, and we ignore those since they send us to a state that is still in t_1 (as that was the epsilon closure of . . .). When those epsilons finish, we find the second "true" transition, y_{j+1}. We now have a previous state $r_j \in t_1$, and the next state is $r_{j+1} \in \delta_N(r_j, y_{j+1})$, which means that r_{j+1} is one of the states in the set $t_2 = \delta_D(t_j, y_{j+1})$.

We continue in this fashion, moving to a new t_k every time we encounter a non-epsilon y, and that $t_k = \delta_D(t_{k-1}, y)$ is exactly the transition in M_D from the previous state and on input that non-epsilon element. All epsilons keep us at the same t_k, since that contained all the epsilon transitions of its elements. Finally, we can write the string w as the concatenation of those non-epsilon elements, as the only things we take out are epsilons.

Lastly, let's look at the last t. It contains the last r_k, and that is in F_N. By the definition of F_D, we have that $t \in F_D$. Therefore the string w is accepted by the DFA M_D.

Now we discuss the converse. Suppose we start with a string $w = z_1 z_2 \cdots z_k$ and DFA states $t_0, t_1, \ldots, t_k$ such that:

- $t_0 = q_0^{(D)}$.
- $t_{i+1} = \delta_D(t_i, z_{i+1})$.
- $t_k \in F_D$.

We need to construct a "path" in the NFA. We will actually work backwards, starting from the end.

Since $t_k \in F_D$, we know that $t_k \cap F_N \neq \emptyset$. So there is some state r_k in t_k that is also in F_N, i.e. a final state of M_N. Now we consider t_{k-1}. What we do know is that:

$$t_k = \delta_D(t_{k-1}, z_k) = E\left(\bigcup_{q \in t_{k-1}} \delta_N(q, z_k)\right)$$

What this literally means is that there is some state $r_{k-1} \in t_{k-1}$ so that $\delta_N(r_{k-1}, z_k)$ followed by epsilon transitions would lead us to r_k. So the end of our string, written to match the NFA, would look like $z_{k-1}\epsilon\epsilon\cdots\epsilon z_k$, and there would be a corresponding chain of states from r_{k-1} to r_k to match it.

We proceed in the same manner to find r_{k-2}, r_{k-3} and so on. At the end we arrive at a state $r_0 \in q_0^{(D)}$, which is within epsilon-transition reach from $q_0^{(N)}$. This would complete the construction of the path.

So this is a remarkable fact, worth repeating:

> A language is regular if and only if it is recognized by an NFA.